MashQL: A Query-by-Diagram Topping SPARQL

Towards Semantic Data Mashups

Mustafa Jarrar
University of Cyprus
mjarrar@cs.ucy.ac.cy

Marios D. Dikaiakos
University of Cyprus
mdd@cs.ucy.ac.cy

ABSTRACT

This article is motivated by the importance of building web *data* mashups. Building on the remarkable success of Web 2.0 mashups, and specially Yahoo Pipes, we generalize the idea of mashups and regard the Internet as a database. Each internet data source is seen as a table, and a mashup is seen as a query on these tables. We assume that web data sources are represented in RDF, and SPARQL is the query language.

We propose a query-by-diagram language called *MashQL*. The goal is to allow people to build data mashups diagrammatically. In the background, MashQL queries are translated into and executed as SPARQL queries. The novelty of MashQL is that it allows querying a data source without any prior understanding of the schema or the structure of this source. Users also do not need any knowledge about RDF/SPARQL to get started.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Query formulation, Information filtering, Retrieval models

General Terms

Languages, Human Factors, Design, Management

Keywords

Query-by-Diagram, Mashups, Query Pipelines, Semantic Web, Data Web, Linked Data, Web 3.0, Web 2.0, RDF, SPARQL

1. Background and Motivation

As this is still an ongoing research, the latest findings can be followed in the evolving technical article [14].

The rapid growth of Web 2.0 content has created a high demand for making this content more reusable. Companies are competing not only on gathering more content and contributions from people, but also on making their content available for using inside their own websites. Many companies such as Google, Yahoo, Microsoft, Amazon, eBay, LinkedIn, and Wikipedia, have made their content publicly accessible through APIs. People are encouraged to make their own applications and profit based on others' content. For example, one can build a program to access the content of the Craigslist real-state database to find apartments in certain area, mix this content with location information from Google Maps, and provide a new web service that was not originally provided by either source. Another web service can be created to find the events happening in a city, by integrating the content of several event databases (such as Upcoming, and Google Base), mix the results with relevant photos from Flickr, and render the final results on Yahoo Maps. Web applications that consume content originated from third parties and retrieved via a public interface or API are called *Mashups*.

ONISW'08, October 30, 2008, Napa Valley, California, USA.

To expose the massive amount of public content and to allow people to build mashups easily, several *mashup editors* have been launched, including Google Mashup, Microsoft's Popfly, IBM's Smash, Yahoo Pipes, and few others. Yahoo Pipes have received the greatest attention thanks to their simplicity. Yahoo Pipes allow people to combine different data sources into mashups, in a graphical and user-friendly way without having to write code. Yahoo Pipes generalize *the idea of the mashup, providing a drag and drop editor that … can easily fetch data from any data source providing an RSS, Atom or RDF feed, extract the data the user wants, combine it with data from other sources, apply various built-in filters, and have the output directed to a web page or to other users' pipes*" [17]. Some people consider Yahoo Pipes to be "*a milestone in the history of the internet"* [17]. The most interesting part in this, is that a user does not really need to have technical experience to get started with Yahoo Pipes.

However, the limitation of Yahoo Pipes and other mashup editors is that they focus only on *web feeds* that can be published in RSS, Atoms, or RDF-feeds. These formats are capable of only representing *news items*; they are not capable of representing *data items* retrieved from the so-called Deep Web and encoded in RDF and XML. Currently, the development of *data mashups* requires extensive programming skills.

To build on the remarkable success of Web 2.0 mashups, we propose to regard mashups as data queries. In other words, we would like to generalize the idea of Web 2.0 mashups and regard the Internet as a database, where each data source is seen as a table, and a mashup is seen as a query. Querying Internet data sources should be as easy as querying database tables. This view is not limited to mashing up Web 2.0 feeds, but can be generalized to data retrieval and integration scenarios.

Assuming the Internet data is represented in RDF, querying this data can be done using SPARQL [21], the RDF query language. SPARQL is a recent recommendation by the W3C. It allows one to query remote RDF resources, in a manner similar to the querying of databases using SQL. A SPARQL query is a set of

graph patterns; any data triple matching these patterns is added to the query results. An example shown in Figure 1 retrieves Hacker's articles published after 2000 from two web locations.

http://Site1.com/RDF

```
:a1 :Title "Web 2.0"
:a1 :Author "Hacker B."
:a1 :Year  2007
:a1 :Publisher "Springer"
:a2 :Title "Web 3.0"
:a2 :Author "Smith B."
```

http://Site2.com/RDF

```
:4 :Title "Semantic Web"
:4 :Author "Tom Lara"
:4 :PubYear  2005
:5 :Title "Web services"
:5 :Author "Bob Hacker"
```

Query:

```
PREFIX S1: <http://site1.com/rdf>
PREFIX S2: <http://site1.com/rdf>
SELECT ? ArticleTitle
FROM <http://site1.com/rdf>
FROM <http://site2.com/rdf>
WHERE {
 {{?X S1:Title ?ArticleTitle}UNION
 {?X S2:Title ?ArticleTitle)}
 {?X S1:Author ?X1} UNION {?X S2:Author ?X1}
 {?X S1:PubYear ?X2} UNION {?X S2:Year ?X2}
 FILTER regex(?X1, "^Hacker")
 FILTER (?X2 > 2000)}
```

Results:

ArticleTitle
Web 2.0

Figure 1. An example of a SPARQL query.

Although RDF has been standardized by W3C since 1999 to play the role of a semantically enabled metadata model only recently has it received a special attention from leading companies. For example, Yahoo announced that the next generation of their search engine will understand web semantics through RDF [27]. Several models of RDF (such as RDFa and eRDF, microformats, and standard vocabularies) will also be supported by Yahoo. *MySpace* announced that they are adopting the semantic web technology and that they will use RDF for profile and data portability [16]. *Upcoming* is already publishing their content in microformats and RDFa (which is a new way of annotating XHTML web pages with RDF triples). Furthermore, Oracle 11g supports RDF storage and query. Querying RDF in Oracle is done in a SPARQL-like style. As shown by Oracle in [4], this implementation is scalable. For example, a query with a medium size complexity over 80 Million RDF triples (5.2 GB) takes one or few seconds. This support from the leading companies is indeed accelerating the adoption of RDF as the main metadata language. Therefore, we believe that RDF and SPARQL mashups will be an important trend of web applications in the near future.

The problem is that building data mashups requires high programming skills and intensive efforts. There is no yet an approach to easy access and expose *structured data* on the web. In the case of using RDF and SPARQL, this challenge is complicated even for some IT people [9]. Understanding the structure of an RDF source (in order to formulate a query about it) is a challenging task indeed. Before formulating a query about an RDF source, one needs to know how the data is structured, and what are the labels of the data elements, i.e., the schema. The problem is that RDF data may come without a schema (as shown in Figure 1), or the schema is mixed up with the data, which is difficult to understand. People typically go over the RDF data manually, read, and mentally build a schematic view of data, as well as remember the names of the data elements. This scenario of understanding RDF (which we call *eye parsing*) can only work with toy examples. However, in case of large RDF sources with diverse content, how you would manage to understand the data structure, inter-relationships, and the unwieldy labels of the data elements. Compared with databases, writing an SQL query requires also that the writer understands the underlying database schema; however, there is no database without a schema and such schemas are typically small and manageable. Formulating structured queries in open environments, where data sources may come without schemes or these schemes are very difficult to eye-parse, is a hard challenge, and thus may hamper the whole utility of RDF and SPARQL. In addition, RDF and SPARQL are unwieldy technical languages, and their intuition of representing knowledge in directed labeled graphs and graph patterns [19,20] is not familiar to most IT people. The lessons learned from Yahoo Pipes show that the simplicity of building feed mashups is the key factor behind its success.

This article proposes presents our early research findings on developing data mashups intuitively. We propose a query-by-diagram language called MashQL, which uses SPARQL as a backend query language. It encapsulates the complexity of SPARQL and allows people to query RDF sources intuitively (see Figure 2). In the background, MashQL queries are translated into and executed as SPARQL queries. The novelty of MashQL is that it *allows one to formulate a query over a data source(s) without any prior knowledge about its schema*. MashQL does not also assume any knowledge about RDF or SPARQL to get started. Hence, the average internet user can use MashQL to develop data mashups easily.

The next section overviews the old and new approaches to query formulation. Section 3 presents the intuitions and the basics of MashQL. In section 4 we present three use cases, and in section 5 we discuss the lessons learnt from these cases. Section 6 discusses the implementation issues. Our conclusions and future directions are presented in section 7.

2. Related Work and Contributions

In this section we overview different approaches to query formulation, focusing on the usability of these approaches for non-IT people.

Query-by-form is an old practice; users can fill in and submit a form, where all fields in this form are seen as query variables. This way of data access is simple; however, it is neither flexible nor expressive. For each query, a form needs to be developed, and any change to the query implies changing the form.

Query-by-example allows users to formulate their queries as filling a table [28]. The names of the queried relations and fields are selected first; then users can enter their keywords. Although this approach is claimed to be easy to learn by non-IT people, however, it was not used by such people. In our opinion, this is because users are still required to understand the relational structure, which is difficult for non-IT people.

Conceptual query languages are an alternative approach to query formulation. As many databases are modeled conceptually using EER or ORM diagrams, one can also query these databases starting from those diagrams. Users can select some concepts from a given conceptual diagram, and their selection is automatically translated into SQL queries. This scenario was implemented by several EER-based [5,18] and ORM-based [7,8] approaches. ConQuer [3] is another ORM-based language, but it has some nice features indeed. Instead of starting from a conceptual diagram that may not exist, it starts from the logical schema and converts it into lists of concepts and relations. Users can then drag-drop from these lists to formulate their queries. What users drag-drop become a tree of facts, and this tree is seen as a query. Although this drag-drop scenario is not simple,

however structuring a query as a tree-pattern looks intuitive indeed.

Although conceptual query languages received a considerable amount of research, but none of these languages was used in practice. In our opinion, this is because formulating a query starting from a conceptual diagram is still a difficult task for non-IT people. In addition, the need to query databases at the conceptual level is not an important issue, because a database is a single enterprise's project, and the world of its developers and users is closed. In open worlds such as the Web, structured data is being created and consumed by different users, and the need for a mechanism to mash up and consume this distributed and heterogeneous data easily is a real demand.

In the recent years, we are observing some advanced techniques start to emerge for filtering streams of information, which we call **Query-by-Filter**. For example, Yahoo Pipes does not support any query language; however, the Filter module has some general concepts, which allow people to permit/block items according to a certain set of conditions. One may block any web feed that contains (/doesn't contain/the same as/ LessThan…) a certain keyword in the title of a feed. This way of expressing filters is also used in most email applications for filtering and organizing emails. Google Base allows one to search information in a mixture of query-by-form and query-by-filter manner. Although the flexibility and expressivity of such filters are very limited, if compared to query languages; however, this approach is well understood and being successfully used by non-IT people.

Furthermore, the need for simplified query techniques is receiving a high importance within the semantic web community. Most approaches are proposing to **Visualize Triple Patterns**, see GRQL [2], iSPARQL [10], NITELIGHT [23] and RDFAuthor [22]. The idea is to represent triple patterns graphically as ellipses connected with arrows, so that one would need less programming skills to formulate a query. Other semantic web approaches are suggesting to use visual scripting languages, such as SPARQLMotion [25] and Deri Pipes [26]. Their idea is to allow users to use visual box and lines, however, queries are written in a textual form. We found all of these approaches assume *advanced* knowledge of RDF and SPARQL, thus cannot be used by the casual user.

Please refer to [15] about a usability study on what casual users prefer, which concludes that a query language should be close to natural language and graphically intuitive.

MashQL is a generalization and extension to many aspects of the above approaches, yielding a formal and expressive but yet simple, query-by-diagram language. MashQL inherits some aspects from conceptual queries and query-by-filters. Similar to ConQuer (and somehow LISA-D), MashQL queries are represented as trees, which makes queries easy to understand. Tree branches in MashQL are similar to filtering rules, which makes query formulation as simple as building filters. The look-and-feel of the MashQL is inspired from Yahoo Pipes.

The difference between MashQL and the query-by-filter approaches is that MashQL is a general language for querying any structured data, not only filtering a specific structure of a data stream, as in Yahoo Pipes. In addition, unlike conceptual queries that start from a conceptual or logical database schema, MashQL is fundamentally different as it assumes that it is not necessary for the queried data sources to have a schema at all.

3. The Basics of MashQL

The goal of MashQL it to allow people to query and mash up web data sources easily. In the background, MashQL queries are automatically translated into and executed as SPARQL queries. People can build data mashups without having to know the underlying structure or technical details of the data sources. Figure 2 shows a MashQL query that is equivalent to the SPARQL query in Figure 1. The first module specifies the query input, while the second MashQL module specifies the query body. The output of this query can be piped into a third module (not shown here), which renders the results into a certain format (such as HTML or XML), or as RDF input to other MashQL queries.

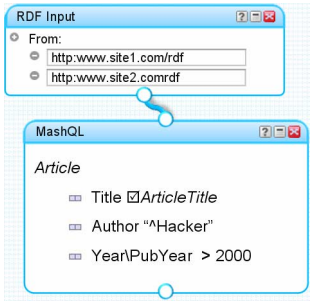

Figure 2. An example of MashQL query.

The intuition of MashQL is described as the following: Each MashQL query is seen as a tree. The root of this tree is called the *query subject* (e.g. Article), which is the subject matter being inquired. Each branch of the tree is called a *query restriction* and is used to restrict a certain property of the query subject. Branches can be expanded to allow sub trees (called *query paths*), which enable one to navigate the underlying data sources (see Figure 3). This query retrieves the recent articles from Cyprus, i.e. every article that is written by an author, who has an address, this address has a country called Cyprus, and the article is published after 2000.


```
PREFIX …
SELECT ?ArticleTitle, ?Institute
FROM …
WHERE {
 ?Article :Title ?ArticleTitle
 ?Article :Author ?X1
 ?X1 :Address ?X2
 ?X2 :Country ?X3
 OPTIONAL{?X1 :Affiliation ?Institute}
 ?Article :Year ?X4
 FILTER (?X3 = "Cyprus")
 FILTER (?X4 > 2000)}
```

Figure 3. Query paths (/sub trees) in MashQL.

Formulating MashQL queries is designed to be an interactive process, in which *the complexity and the responsibility of understanding data structures are moved from the user to the query editor*. Users only use drop-down lists to express their queries. While interacting with the query editor, the editor performs some *background queries* and dynamically generates these lists. In what follows, we describe these background queries.

After a user selects the dataset in the RDF Input module, formulating a MashQL query is done by first selecting the query

subject, which is offered through a drop-down list generated from (the union of all subject and object identifiers in the dataset), no matter whether an identifier represents an instance or a type. Users can also choose not to select from the list and introduce their own subject label. In this case, the subject is seen as a variable and displayed in *italic,* which means any data item[1].

To add a restriction on the chosen subject, a (list of the possible properties for this subject) is dynamically generated. For example, given the data in Figure 1, if a user chooses *a1* as a subject, the list of the *a1*'s properties will be {Title, Author, Publisher, Year}; if the subject is a variable, the list will be the set of all properties in the dataset. Users can also choose whether this property is required, optional, or unbound. If a property is prefixed with "maybe" this property is considered optional (see Figure 3), if it is prefixed with "without" it is considered unbound, and if it is not prefixed then it is required.

Users may then choose an object filter such as (Equals, Contains, Doesn't contain, OneOf, Between, MoreThan, Not, etc.), or may select an object identifier from a list, which is generated from (the set of the possible objects, depending on the previously chosen subject and predicate). Furthermore, users can also click on the restriction icon to expand the tree, i.e., declare a query path as shown in Figure 3. The symbol ☑ can be used before subject, property, or object variables to indicate that this variable will be returned in the results. For example, the results of the above query are a one-column table that contains the list of all retrieved titles.

MashQL supports several other constructs that are not presented in this paper, such as union (denoted as "\") between objects, predicates, subjects, and queries; as well as, a type operator ("a"), reverse predicates, OneOf, datatype and language tags, and many object filters. The full syntax of MashQL, formal semantics, and the mapping into SPARQL are being completed and can be found in our evolving technical report [14].

The trade-off between expressivity and simplicity in MashQL is achieved by making *technical variables and namespaces to be implicit*, and through *the tree structure of MashQL* queries, which is close to the intuition people use in their natural language communication. For example, the query path shown in Figure 3 means, retrieve the article that has an Author x_1, and x_1 has an address x_2, and x_2 has a country x_4, and x_4 equals "Cyprus". Furthermore, suppose you would like to ask; "Give me the list of all stores that sell parts of the iPhone mobile, and that are located in Brussels"; or, "Which cinemas are located in Brussels, offer a movie called 'Fahrenheit' and will be played between 20:00 and 23:00". Apart from some terms (such as: give me the list of all, which, who, that are), all of these inquiries can be directly converted into MashQL queries. Hence, MashQL can be used by both the average internet users and IT professionals to create data mashups intuitively and as expressive as SPARQL.

Remark: MashQL supports some novel interface issues that are lengthy or difficult to illustrate here, especially the edit-and-verbalize modes. For example, when a user clicks on a restriction, it gets the *editing mode* and all other restrictions get the *verbalize mode* (i.e., all boxes and lists are made invisible, but the verbalization of their content is generated and displayed instead, as shown in all figures). This does not only make the query formulation process even easier and joyful, but more importantly, from a methodology viewpoint it makes the readability of the queries closer to natural language, by which users are guided to achieve what they intended to query. Similarly, when two predicates originating from different sources have the same label, their namespaces are hidden and one of them is displayed, unless the user decided not so.

Furthermore, similar to the idea of pipelining web feeds in Yahoo Pipes, or pipelining software processes in Unix, MashQL allows queries to be pipelined. The idea is that the output of a query is used as input to another (see Figure 4). In this way, people who develop data mashups can reuse the output of others' mashups. For example, person A builds a mashup to query all articles published by Springer (Q_1); Person B builds a mashup to query all articles published by ACM (Q_2); Person C filters the results of Q_1 and Q_2 to get only the articles published in 1997 and by Italian authors. Query pipelining is a built-in concept in MashQL, not only as a user interface issue. As Figure 4 shows, depending on the structure of the queries and how they connect to each other, MashQL generates either a `SELECT` or a `CONSTRUCT` statement.

4. Use Cases

To demonstrate the utility of MashQL in solving some real-life problems, and to learn what are the important constructs to include or exclude, we developed a number of use cases corresponding to "real-life" application scenarios [14]. Here, we briefly illustrate three scenarios. In the section, we discuss the lessons learnt from these cases.

4.1 Use case: Job Seeking

Bob has a PhD in bioinformatics. He is looking for a full-time, well paid, and research-oriented job in some European countries. He spent an enormous amount of time searching different job portals, each time trying many keywords and filters. Instead, Bob used MashQL to find the job that meets his specific preferences. Figure 4 shows Bob's queries on Google Base and on Jobs.ac.uk. First, he visited Google Base and performed a keyword search (bioinformatics OR "computational biology" OR "systems biology" OR e-health); he copied the link of the retrieved results from Google into the RDFInput module[2]; and then created a MashQL query on these results. He performed a similar task to query Jobs.ac.uk. The third MashQL module in Figure 4, mixes the results of the above two queries and filters them based on location preferences (provided in the UserInput module). The SPRQAL equivalent to Bob's MashQL query is shown in Figure 5.

Notice that we translate MashQL queries that pipe each other using "`CONSTRUCT *`", which is not part of the current SPARQL standard. However, this construct is one of the top proposed extensions to SPARQL (see [24]). Otherwise, if this construct will not be included in the next version of SPARQL, our translation will return every triple involved in the query patterns.

[1] The default value for a subject (in case a user does not select from the offered list or introduce his own label) is the variable "*Everything*".

[2] We assume that both Google and Jobs.ac.uk render their search results in RDFa (i.e. the RDF triples are embedded in HTML), as many companies started to do nowadays. However, Bob can also use a third party's service (e.g. triplify.org) to extract triples from HTML pages.

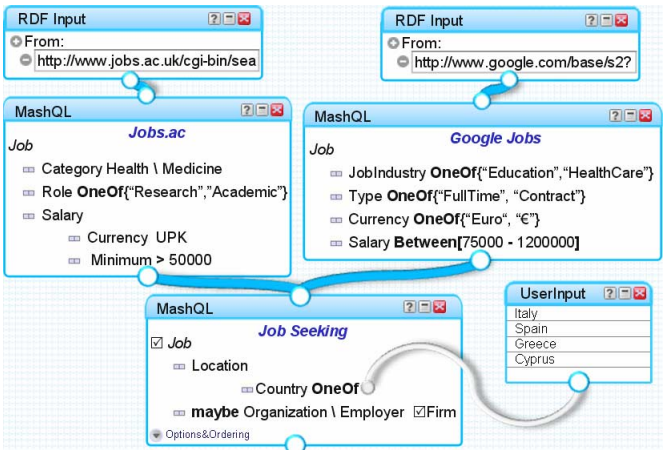

Figure 4. Bob's MashQL Queries.

```
…
CONSTRUCT *
WHERE {
{{?Job :Category :Health}UNION
{?Job :Category :Medicine}}
?Job :Role ?X1.
?Job :Salary ?X2.
?X2 :Currency :UPK.
?X2 :Minimun ?X3.
FILTER(?X1="Research" ||
       ?X1="Academic")
FILTER (?X3 > 50000) }
```

```
…
CONSTRUCT *
WHERE{?Job :JobIndustry ?X1.
      ?Job :Type ?X2.
      ?Job :Currency ?X3.
      ?Job :Salary ?X4.
FILTER(?X1="Education"||
       ?X1="HealthCare")
FILTER(?X2="Full-Time"||
       ?X2="Fulltime")||
       ?X2="Contract")
FILTER(?X3="^Euro"||
       ?X3="^€")
FILTER(?X4>=75000||
       ?X4<=120000)}
```

```
…
SELECT ?Job ?Firm
WHERE {?Job :Location ?X1. ?X1 :Country ?X2.
       FILTER (?X2="Italy"||?X2="Spain")||
              ?X2="Greece"||?X2="Cyprus")}
       OPTIONAL{{?job :Organization ?Firm}UNION
                {?job :Employer ?Firm}}
```

Figure 5. The SPARQL translation of the MashQL queries in Figure 4.

4.2 Use case: eHealth Research

Alice is a PhD student in biology, she wants to search an online eHealth database, to know what causes prostate cancer most. First, she wrote a simple query to find all patients that certainly have a prostate cancer i.e. every person whose prostate biopsy test is positive. Suppose she found 3500 cases. Then she started to add and remove restrictions to this query. Her goal was to reach the closest number to 3500, with a maximum number of relevant restrictions. Alice found that the restrictions shown in Figure 6 are the indicators for 90% of the people who had cancer.

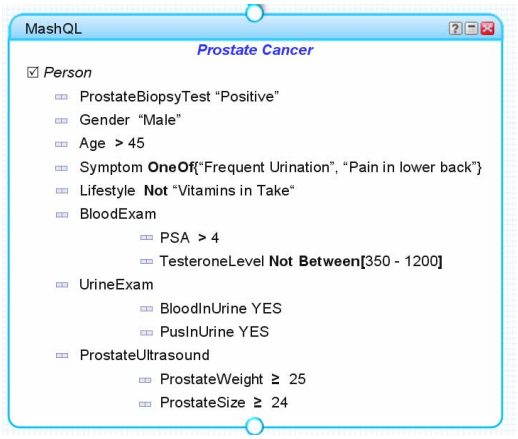

Figure 6. Alice's research query.

4.3 Use case: Car Rental

This use case demonstrates a different and an offline use of MashQL for declaring business rules in an auditing application.

The government usually audits whether car rental companies comply with the local regulations. All companies have to open their databases for auditing. The auditors visit companies irregularly, and run a set of queries (called *auditing queries*) to discover violations. As each company has different database design and vocabulary, the auditors have to analyze and understand the database schema, and write the auditing queries each time from scratch. The government decided to introduce the new semantic technology. They built a wrapper that connects to any database and automatically converts this into one large RDF table, this takes few minutes to complete. See a sample of such data in Figure 7. The auditors then use the MashQL editor to write and execute their set of auditing queries. Figure 8 shows three examples of such auditing queries.

The first query checks whether there is a car rental that occurred during a period that this car was not insured. The result was {<:Rental1>} as the car was not insured during the first 3 days of the rental. The second query checks whether there is a car rental occurred for a customer who does not have a driving license; the result is empty. The third query checks whether there is a car rental occurred for a customer whose driving license does not authorize him to drive that type of car ; the result is {<:Rental2>}, because the license of Customer2 is B, while the category of the car he rented requires license C or higher.

<http://localhost/Company1>

:Vehicle1 rdf:type :Vehicle	
:Vehicle1 :PlateNumber "ABC323"	
:Vehicle1 :VehicleCategory "C"	:Customer1 :LicenseNumber "8723"
:Vehicle1 :Insurance :InsContract1	:Customer1 :LicenseType "B"
:Vehicle2 rdf:type :Vehicle	:Customer2 rdf:type :Customer
:Vehicle2 :PlateNumber "BDC987"	:Customer2 :CustomerID "3456"
:Vehicle2 :VehicleCategory "B"	:Customer2 :LicenseNumber "8723"
:Vehicle2 :Insurance :InsContract2	:Customer2:LicenseType "B"
:InsContract1 rdf:type :VehicleInsurance	:Rental1 rdf:type :Rental
:InsContract1 :InsuranceType "Full"	:Rental1 :Vehicle :Vehicle2
:InsContract1 :InsuranceSDate 23-11-2007	:Rental1 :Customer :Customer1
:InsContract1 :InsuranceEDate 22-11-2008	:Rental1 :StartOn 20-11-2007
:InsContract2 rdf:type :VehicleInsurance	:Rental1 :EndsOn 25-11-2007
:InsContract2 :InsuranceType "Full"	:Rental2 rdf:type :Rental
:InsContract2 :InsuranceSDate 10-04-2007	:Rental2 :Vehicle :Vehicle1
:InsContract2 :InsuranceEDate 11-04-2008	:Rental2 :Customer :Customer2
:Customer1 rdf:type :Customer	:Rental2 :StartsOn 15-02-2008
:Customer1 :CustomerID "6783"	:Rental2 :EndsOn 16-02-2008

Figure 7. Sample of RDF for a car rental company.

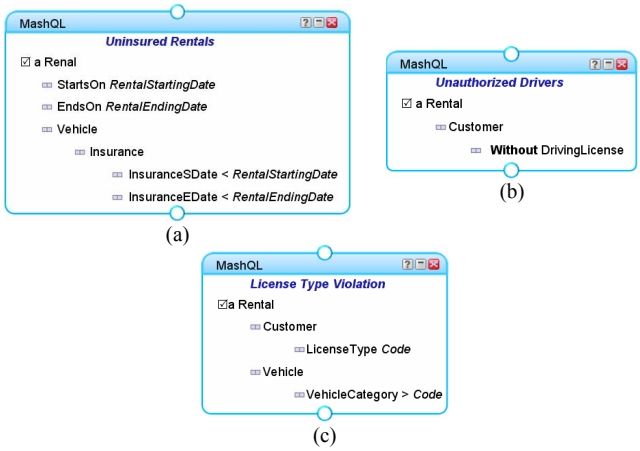

Figure 8. Examples of Auditing Queries.

5. Discussion and Lessons Learnt

5.1 Coverage and Elegance

We have learned from these use cases that some MashQL constructs (especially the OneOf, Between, and the Union "\") are useful to have in the language, because they were used in most cases. On the other hand, we found that some important constructs are missing, specially the aggregation functions. Thus, we decided to extend our approach and use Oracle's SPARQL, instead of only the W3C's SPARQL standard. Oracle's SPARQL inherits all functionalities of SQL, including aggregation functions, grouping, among others.

All SPARQL constructs are somehow supported by MashQL; and all MashQL constructs can be easily expressed in SPARQL. Some constructs are mapped directly, but some other constructs require lengthy emulation. This shows indeed that MashQL constructs are more concise and intuitive than SPARQL. For example, SPARQL query scripts containing the union operator require almost twice the size than their MashQL equivalents. The OneOf operator, which is represented as a set of constants in MashQL, is emulated by long SPARQL filters.

The main limitation of MashQL's elegancy is that in case an RDF data source contains unwieldy terms, this may yield inelegant MashQL queries. For example, suppose you want to query this dataset: {<:C1,:eno,AB12345> <:C1,:eid,987665> <:C1,:efname,Bob>}. The technical labels of predicates which are difficult to understand by people that did not create them may yield inelegant or technical queries. However, we believe that if an RDF source is made public for external users, predicate vocabularies will be clear and mostly based on standard RDF vocabularies (such as Dublin Core, FOAF, GeoRSS, etc.), or based on a shared ontology [11].

Furthermore, the use cases also taught us that representing queries visually (as modules connected through pipes) helps non-IT users to visualize and understand the *information flow* in their queries. In other words, this visualization facilitates users to organize and modularize their queries at a high level of abstraction. For example, one may notice that Bob can create only one query and get the same results of the three queries shown in Figure 4. However, instead of building such a complex query, Bob preferred to modularize his queries in this way, as it is easier for him to build and maintain. In short, modularizing queries and piping them in this way is easier to abstract for non-IT experts.

5.2 Performance and Complexity

Suppose one asks how long it takes to execute the query shown in Figure 2. The answer is: exactly the same time the query shown in Figure 1 would take, which is its SPARQL translation. In other words, although users build and view their mashups using MashQL, but MashQL queries are not executed themselves, their SPARQL translation that is executed. The time complexity of translating a MashQL query into SPARQL is neglectable as it takes less than a second. Thus, the complexity of executing a MashQL query is bounded to the complexity and performance of its backend query language, which is SPARQL in our case.

Executing a SPARQL query over a huge RDF dataset is indeed very fast. As shown by Oracle in [4] and AllegroGraph [1], a query with a medium size complexity over hundreds of millions of triples takes only one or few seconds. Oracle has proven indeed that both querying and bulk-loading of RDF data is scalable [4,6]. However, a problem may arise in case of querying remote RDF sources. When a user defines a query over a set of remote sources, as shown in Figure 1, these sources must be transferred and stored locally before executing the query [26]. Hence, the time required to execute a query depends mainly on the amount of the transferred data and the transferring bandwidth. The performance of using SPARQL (and thus MashQL) to mash up large remote sources might be unacceptable in case these sources are very large. However, in most application scenarios in practice (see the use cases in [14]), people would build mashups on acceptable sizes of data sources. For example, querying HTML pages that contain RDFa, small RDF files, or retrieved results from online RDF stores that support SPARQL endpoints, among many other scenarios. As this topic is related to the performance of SPARQL, rather than MashQL, we tackle it separately; see the future work section.

Furthermore, recall that when formulating a MashQL query, there are some background queries that are used to generate drop-down lists (see section 3). These queries should be fast enough to allow efficient query formulation. For example, after specifying the RDF sources to be queried, users can select the query subject, which is offered through a drop-down list that is dynamically generated from the union of the subject and object identifiers in the data sources. Similarly, predicates and objects are selected from dynamic lists (see section 3). In general, for a query with n restrictions, there are at most $(2n+1)$ queries that will be performed in the background, i.e. during the query formulation process. The performance of these queries should be fast: as soon as a user selects from a list and moves the mouse to select from another list, this list should be ready. As discussed above, in case the queried sources are stored locally, the performance of the background queries is indeed fast, as each query takes only one second. However, in case of remote input sources, these sources need to be transferred and stored locally, before the query formulation process starts. Therefore, query formulation in case of large remote sources might be unacceptable. In section 7, we shall discuss our future plans to overcome this challenge, which a SPARQL rather than a MashQL issue.

6. Implementation Specification

We have developed a MashQL markup based on XML. The goal of this markup is to serialize MashQL pipes in a textual and interchangeable format. Hence, one would save, reload, process, and exchange MashQL queries easily. Figure 9 illustrates the markup of the MashQL pipes shown in Figure 2, which is also equivalent to the SPARQL query in Figure 1. As shown in this figure, the markup of the MashQL pipe consists of three main elements: Meta, RDFInput, and Query. The Meta is used to represent metadata about the pipe itself, and the RDFInput represents the sources in an RDFInput module. The Query consists of three sections: Header, Body, and Footer. While the Body serializes mainly query conditions, the Header serializes the input sources, prefixes, and some metadata about the query. The result modifiers, ordering preferences, and output styles are serialized in the Footer part. These parameters are usually not displayed in the main window of a MashQL query, but can be configured as "query options". The XML schema of the complete

MashQL markup is presented and discussed in [14]. This schema is used as the technical specification, (i.e. the reference grammar) for MashQL. Any markup of a MashQL pipe should be valid according to this XML-Schema.

Furthermore, we have also developed a MashQL-to-SPARQL translator, which is a java program. This translator takes the markup of a MashQL query as input and generates its SPARQL equivalence. When a user debugs or executes a MashQL query, the query editor calls the translator to generate its SPARQL equivalent on the fly, and then executes it as a SPARQL query. More details about the translator and other implementation issues can be found also in [14].

```
<Pipe ID="0" xsi:noNamespaceSchemaLocation="MashQL.xsd"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <Meta Content="Title" Name="Bob's Citations"/><Meta Content="Creator" Name="
Hacker"/>

 <RDFInput ID="1" X1="10" Y1="10" X2="50" Y2="110">
  <Source Order="1">
   <Ref>http://www.site1.com/rdf</Ref><LastUpload>2008-06-02T09:30:47.0Z
</LastUpload>
  </Source>
  <Source Order="2">
   <Ref>http://www.site2.com/rdf</Ref><LastUpload>2008-06-02T09:32:47.0Z
</LastUpload>
  </Source>
 </RDFInput>

 <Query ID="0" X1="0" X2="0" Y1="xs:integer" Y2="0" InputModule="1">
  <Header> <Meta Content="Title" Name=""/>
<Prefix Tag="S1" Ref="http://www.site1.com/ "/><Prefix Tag="S2" Ref="http://www.site2.com
"/>
  </Header>

  <Body>
   <Subject Name="Article"  Type="Variable" isRetrun="false"/>
   <Restriction Prefix="">
    <Predicate Name="S1:Title" Type="Identifier" isReturn="false" />
    <Predicate Name="S2:Title" Type="Identifier" isReturn="false" />
    <Object Name="ArticleTitle" Type="Variable" isReturn="true" /> </Restriction>
   <Restriction Prefix="">
    <Predicate Name="S1:Author" Type="Identifier" isReturn="false" />
    <Predicate Name="S2:Author" Type="Identifier" isReturn="false" />
    <Object Name="X1" Type="Variable" isReturn="false" />
    <ObjectFilter xsi:type="Contains" Value="^Hacker" Type="Variable" Language=""
DataType="" />
   </Restriction>
   <Restriction Prefix="">
    <Predicate Name="S2;Year" Type="Identifier" isReturn="false" />
    <Predicate Name="S2:PubYear" Type="Identifier" isReturn="false" />
    <Object Name="X2" Type="Variable" isReturn="false" />
    <ObjectFilter xsi:type="MoreThan" Value="2000" Type="Constant" DataType="" />
  </Restriction>
  </Body>

  <Footer>
   <Order><Variable Name="" Direction=""/></Order>
   <Modifier Limit="" Offset="" Duplication=""/>
   <Output Stylesheet="" Format=""/>
  </Footer>
 </Query>
</Pipe>
```

Figure 9. MashQL Markup in XML.

MashQL can generally be implemented by online mashup editors (similar to or as an extension to Yahoo Pipes), or as a query interface for online RDF datasets (e.g., Freebase, DBpedia, or DBLP). It can be implemented also as a query plug-in to offline RDF stores (e.g., AllegroGraph or Oracle); or it can be used to filter metadata streams in, e.g., iTunes, jobs.ac.uk, eBay, or Upcoming.

Although this article focuses on using MashQL for querying RDF data sources using SPARQL, however, MashQL can be similarly used for querying relational databases or XML documents. In this case, one needs to either develop a stylesheet that translates MashQL markups into SQL, XQuery, or any preferred backend query language; or maybe map the dataset (e.g. using views) into an RDF-like model.

As we have discussed earlier, our implementation prototype does not only generate the W3C's standard SPARQL, but being extended to also generate Oracle's SPARQL [4], as it supports aggregation and grouping.

7. Conclusions and Future Work

In this article, we have proposed a query-by-diagram language in order to allow building data mashups intuitively. Not only MashQL is user-friendly for non-IT people, but also it allows querying (and navigating) RDF data sources without having to know the schema or the technical details of these sources. We have demonstrated the use of MashQL using different use cases, and discussed the lessons learned regarding the elegancy, coverage, and performance of MashQL. As we have discussed and demonstrated, MashQL is not merely an interface of SPARQL. Although it can be used as such, but it can be used also as a general query language by its own. In addition, MashQL can be used also for filtering metadata streams. In this article, we focus on using the W3C's SPARQL standard as the backend query language.

However, we plan to also translate MashQL into Oracle's SPARQL. This translation is being implemented as an AJAX web-based plug-in to Oracle 11g. Choosing Oracle is not only because of its scalability, but also because Oracle's SPARQL inherits all functionalities of SQL, including aggregation and grouping functions, which are not supported in the standard SPARQL. Furthermore, as querying large remote sources using SPARQL matters (see our discussion in the section 5.2), we are developing a framework called Semantic Web Pipes. This framework extends MashQL and allows caching remote sources, materializing query result, distributing queries, publishing, and discovery of queries, among other issues. Last but yet important, we also plan to extract schemas from RDF sources and depict these schemas using ORM and based on [12,13], which would be an extra offline support for MashQL users.

The complete syntax of MashQL, its formal semantics, implementation, and evaluation, are being finalized and can be followed in the evolving technical article [14].

Acknowledgement

We are indebted to George Pallis for his valuable comments and feedback on the early drafts of this paper. This research is partially supported by the SEARCHiN project (FP6-042467, Marie Curie Actions).

References

[1] AllegroGraph http://www.franz.com/products/allegrograph (May 2008)

[2] Athanasis, N., Christophides, V., and Kotzinos, D.: Generating On the Fly Queries for the Semantic Web. In Proceedings of the ISWC. LNCS, Springer. 2004.

[3] Bloesch, A. and Halpin, T.: Conceptual Queries using ConQuer–II. In Proceedings of the ER. LNCS, Springer. 1997.

[4] Chong, E., Das, S., Eadon, G., and Srinivasan, J.: An efficient SQL-based RDF querying scheme. In Proceedings of VLDB. LNCS, Springer. 2005

[5] Czejdo, B., Elmasri, R., Rusinkiewicz, M., and Embley, D.: An algebraic language for graphical query formulation using an EER model. In Proceedings of the ACM Conference on Computer Science. 1987.

[6] Das, S., Chong, E., Wu, Z., Annamalai, M., and Srinivasan, J.: A Scalable Scheme for Bulk Loading Large RDF Graphs into Oracle. In Proceedings of the ICDE. ACM. 2008.

[7] De Troyer, O., Meersman, R., and Verlinden, P.: RIDL on the CRIS Case: A Workbench for NIAM. In Proceedings of the IFIP.8.1 Conference. 1988.

[8] Hofstede, A., Proper, H., and van der Weide, T.: Computer Supported Query Formulation in an Evolving Context. In Proceedings of the ADC. 1995.

[9] Iskold, A.: Semantic Web: Difficulties with the Classic Approach. The ReadWriteWeb magazine. Sep. 19, 2007

[10] iSPARQL http://demo.openlinksw.com/isparql (May 2008)

[11] Jarrar, M.: Towards Methodological Principles for Ontology Engineering. PhD Thesis. Vrije Universiteit Brussel. May 2005

[12] Jarrar, M.: Towards Automated Reasoning on ORM Schemes -Mapping ORM into the DLRidf Description Logic. In Proceedings of the ER. LNCS, Springer. 2007.

[13] Jarrar, M.: Mapping ORM into the SHOIN/OWL Description Logic -Towards a Methodological and Expressive Graphical Notation for Ontology Engineering. In Proceedings of the OTM'07 Workshops. LNCS, Springer. 2007

[14] Jarrar, M., and Dikaiakos, M. D.: MashQL: A Query-By-Diagram Language for Data Mashups. Technical Article (No. TAR200805). University of Cyprus, 2008 http://www.jarrar.info/publications/JD08.pdf

[15] Kaufmann, E., and Bernstein, A.: How Useful Are Natural Language Interfaces to the Semantic Web for Casual End-Users. In Proceedings of the ISWC. LNCS, Springer. 2007.

[16] O'Donoghue, J.: MySpace joins the 'semantic' web. The Web User online magazine. May 9, 2008

[17] O'Reilly, T.: http://radar.oreilly.com/archives/ 2007/02/pipes-and-filters-for-the-inte.html (May 2008)

[18] Parent, C., and Spaccapietra, S.: About Complex Entities, Complex Objects and Object-Oriented Data Models. In Proceedings of the IFIP 8.1 conference. 1989

[19] Perez, J., Arenas, M., and Gutierrez, C.: Semantics and Complexity of SPARQL. In Proceedings of the ISWC. LNCS, Springer. 2006

[20] Polleres, A.: From SPARQL to rules (and back). In proceedings of the WWW. ACM. 2007

[21] Prud'hommeaux, E. (ed.): SPARQL Query Language for RDF, W3C Working Draft, 4 Oct. 2006

[22] RFAuthor (May 2008)

http://rdfweb.org/people/damian/2001/10/RDFAuthor

[23] Russell, A., Smart, R., Braines, D., and Shadbolt, R.: NITELIGHT: A Graphical Tool for Semantic Query Construction. In Proceedings of the Semantic Web User Interaction Workshop (SWUI). 2008.

[24] SPARQL Extensions (March 2008) http://esw.w3.org/topic/SPARQL/Extensions?highlight=%28sparql%29

[25] SPARQLMotion http://www.topquadrant.com/sparqlmotion (May 2008)

[26] Tummarello, G., Polleres, A., and Morbidoni, C.: Who the FOAF knows Alice? A needed step toward Semantic Web Pipes. In Proceedings of the ISWC Workshops. 2007

[27] Yahoo Blog (Mar.08) http://www.ysearchblog.com/ archives/ 000527.html

[28] Zloof, M.: Query-by-Example: A Data Base Language. IBM Systems Journal, 16(4). 1977